Parallel local search for the time-constrained traveling salesman problem

G.A.P. Kindervater
Erasmus University, Rotterdam

J.K. Lenstra
Eindhoven University of Technology
Centre for Mathematics and Computer Science (CWI), Amsterdam

M.W.P. Savelsbergh
Eindhoven University of Technology
Georgia Institute of Technology, Atlanta

In the time-constrained TSP, each city has to be visited within a given time interval. Such 'time windows' often occur in practice. When practical vehicle routing problems are solved in an interactive setting, one needs algorithms for the time-constrained TSP that combine a low running time with a high solution quality. Local search seems a natural approach. It is not obvious, however, how local search for the TSP has to be implemented so as to handle time windows efficiently. This is particularly true when parallel computer architectures are available. We consider these questions.

Note: This paper will appear in 'Twenty-five years of operations research in the Netherlands: papers dedicated to Gijs de Leve', edited by Jan Karel Lenstra, Henk Tijms and Ton Volgenant (CWI Tract 70, Centre for Mathematics and Computer Science, Amsterdam, 1990).

1. Introduction

On May 2, 1969, Professor Gijs de Leve showed his newly-appointed assistant around in the Mathematical Centre, then located in an old school building. 'Here is our library,' he said. 'And this is how you do research. You just pick up a journal, and - well, there isn't any Markov programming here, but this may interest you.' The journal was a recent issue of *Operations Research*, and the paper was Bellmore and Nemhauser's survey of the traveling salesman problem [Bellmore & Nemhauser, 1968].

This was neither the first nor the last time that De Leve put someone on the track of the traveling salesman. As a result, the TSP has always occupied a central position in the research in combinatorial optimization at the University of Amsterdam and at the Mathematical Centre. This has led to a long list of publications, which probably starts with the survey by Tijdeman [1968]. It includes De Leve's own elegant improvement of the assignment bound [Jonker, De Leve, Van der Velde & Volgenant, 1980] as well as the impressive computational work of Jonker [1986] and Volgenant [1987]. The latest additions focus on the availability of new computer architectures for interactive and parallel computing and their consequences for the TSP.

In this contribution, we review some of this recent work. We give a nontechnical summary in Section 2. Sections 3-7 provide more detail; most of this material is adapted from Martin Savelsbergh's dissertation on interactive vehicle routing [Savelsbergh, 1988] and Gerard Kindervater's dissertation on parallel combinatorial computing [Kindervater, 1989].

Report EUR-CS-89-07
Erasmus University, Department of Computer Science
P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

2. Nontechnical summary

2.1. Theory versus practice

The theory of operations research is concerned with the investigation of a broad class of mathematical models that are somehow inspired by practical decision situations, and with the design and analysis of algorithms for their solution. The practice of operations research is an even broader and considerably less scientific occupation. A huge and ever growing pile of literature is devoted to the tension between theory and practice and to the inadequacy of the mathematical models and methods in giving real-world solutions to real-world problems.

For the benefit of the reader, we summarize this literature in one paragraph. The first observation is that decision problems tend to be both *soft* and *hard*. At the practical side, the decision situation is usually ill-defined and the quality of a decision, as expressed by notions like feasibility and optimality, is an imprecise concept. Feasibility requirements may be loose rather than strict, and tradeoffs between optimality criteria are often not explicitly known but carried implicitly in the value judgement of the decision maker. At the mathematical side, any reasonable abstraction of the decision situation is likely to be computationally intractable in a well-defined sense. The second observation is that no solution can be better than the model to which it provides an answer. While the construction of models that are both realistic and tractable is a delicate affair, the implementation of solution procedures and their results in practice is far more difficult. For applications of operations research, the modeling stage is a minor obstacle in comparison to the implementation stage.

One way out of these complications, which has been much advocated, is to create a so-called *gap* between theory and practice and to try to fill it up with the literature on operations research. Another approach, which yields more mutual benefits, goes under the name of *man-machine interaction*. The idea is that man and machine each have their given and complementary capabilities. Human problem solving is empirical by nature, based on generalization, insight, and experience. Automated problem solving is normative and proceeds by the efficient application of general rules in specific situations. An *interactive planning system* combines the strengths of both approaches. Roughly speaking, the planner is in charge of the global problem aspects and takes care of all kinds of ad hoc constraints, and the computer performs the routine work, such as the manipulation and representation of data and the solution of detailed subproblems. (The reader should note the contrast with *artificial intelligence*, which is concerned with the automation of tasks that are better done by human beings.) We refer to Anthonisse, Lenstra & Savelsbergh [1989] for a further elaboration on the functional and technical characteristics of these types of systems.

2.2. Local search

The emergence of interactive planning systems has reinforced the need for algorithms that can handle problems of a realistic size and give solutions of a reasonable quality in a reasonable amount of time. One often employs some form of local search of the solution space. Although theoretical results on the performance of local search algorithms are scarce and mostly negative, it is generally acknowledged that their empirical behavior is excellent. In addition to being *effective* and *efficient*, local search is also *robust* and *easy to program*. That is, a local search method for a certain model is usually readily adapted to handle minor variations of the model, and developing a computer code requires much less effort than in the case of highly structured optimization algorithms or tailored approximation techniques.

Local search owes this flexibility and simplicity to the fact that it proceeds on the basis of relatively little information about the problem under consideration. One only has to specify an *initial feasible solution* and fast subroutines that, given a feasible solution, compute its *cost* (i.e., the value of the objective function) and its *neighborhood* (i.e., a set of feasible solutions that are in some sense close to it). Given a starting solution, its neighborhood is searched for a solution of lower cost. If such a solution exists, it becomes the new starting point and the search continues. Otherwise, a local optimum relative to the neighborhood definition has been found.

This heuristic solution approach enjoys an increasing popularity. Many variants have recently been proposed, such as simulated annealing, tabu search, neural nets, and genetic algorithms. It is not our purpose to discuss this class of so-called *homeopathic algorithms* [Van Hee, 1989]. Rather, we will consider a plain and simple local search method for the TSP and examine its implementation when time constraints are added to the model.

2.3. Local search for the TSP

Like so many other approaches in combinatorial optimization, local search was first seriously investigated in the context of the TSP. Lin [1965] calls a traveling salesman tour *k-optimal* when it cannot be improved by replacing j of its edges by j other edges, for any $j \leq k$. It is not known whether, for any fixed value of $k \geq 2$, a k-optimal tour can be *generated* in polynomial time. However, it is trivial to observe that the k-optimality of a given tour through n cities can be *verified* in $O(n^k)$ time: there are $\binom{n}{k}$ ways to delete k edges; for each of these, there is a constant number of candidate improvements (where the constant depends on k); and each of these candidates can be evaluated in constant time. For example, if $k = 2$, two edges are replaced by two other edges, and only four cost coefficients have to be checked in order to compute the length of the new tour.

Now suppose that each city has its own time window during which it must be visited, and again consider the case $k = 2$. If two edges are replaced by two other edges, then a certain segment of the tour will be traversed in the opposite direction. In addition to the test for improvement, there is now also a test for feasibility with respect to the time windows. This takes time proportional to the length of the reversed segment. In general, a straightforward implementation of the algorithm requires linear rather than constant time for evaluating a single k-exchange and thereby $\Theta(n^{k+1})$ time for verifying the k-optimality of a tour. We will present a way to avoid this additional factor of n and to verify k-optimality for the time-constrained TSP in $O(n^k)$ time.

2.4. Serial and parallel computing

So far, we have implicitly assumed that our algorithms were to be executed on a traditional computer, which performs at most one computation at a time. An algorithm for a given problem is *likable* if the number of computations involved is bounded by a polynomial function of the problem size, and the algorithm is *more likable* if the degree of the polynomial is lower. Thus, we do not know if there is a likable algorithm for generating a k-optimal tour. However, such an algorithm does exist for verifying k-optimality of a given tour, and we like our $O(n^k)$ approach better that the obvious $O(n^{k+1})$ implementation.

Now suppose that we have a computer that can perform a number of operations in parallel. Such a computer has a greater processing power than a serial one. This is especially important in the context of man-machine interaction, where the user expects fast answers in real time.

More specifically, assume that we have an unbounded number of processors that operate in parallel and communicate with each other in constant time. Consider, as an example, the simple problem of finding the maximum of n numbers $a_1, a_2, \ldots, a_n$. At the first stage, one processor takes the maximum of a_1 and a_2, another processor takes the maximum of a_3 and a_4, and so on. At the second stage, about $n/2$ numbers are left, and again pairwise maxima are taken. So it continues. After $\lceil \log n \rceil$ stages, we have the overall maximum. (All logarithms in this paper are to the base 2.) It follows that the problem is solvable in logarithmic time on a linear number of processors and that, in order to achieve this, each processor needs to know only a small fraction of the entire problem instance. Indeed, if a problem of size n is solved in $\log n$ time, no single processor is able to read all of the problem data. It appears that, when we can compute in parallel, we can find algorithms that are *more than likable*.

2.5. Parallel local search for the TSP

We have explained that the maximum of n numbers can be found by $n/2$ processors in $\log n$ time. Similarly, the k-optimality of a tour through n cities can be verified by $O(n^k)$ processors in $O(\log n)$ time: each processor evaluates a single k-exchange in constant time, and the best of these is selected in logarithmic time. In both cases, it is not hard to reduce the number of processors involved by a factor of $\log n$. Hence, for the TSP, $O(n^k/\log n)$ processors do in time $O(\log n)$ what a single processor can do in time $O(n^k)$. We thus achieve a *perfect speedup*.

When time constraints are added, complications occur. Evaluating a single k-exchange seems to be a serial process, but it is not too hard to design a parallel implementation that requires logarithmic time and a linear number of processors. This leads to an algorithm for verifying k-optimality in $O(\log n)$ time using $O(n^{k+1}/\log n)$ processors. Further improvements are possible, and we can save a factor of n in the number of processors, again achieving a perfect speedup.

2.6. Yet another summary

Section 3 gives a brief and informal introduction into the relevant concepts of complexity theory. Sections 4 and 5 discuss serial and parallel local search for the unconstrained TSP, respectively; this material is relatively straightforward. Section 6 presents our implementation of serial local search for the time-constrained TSP, and Section 7 deals with the parallel case.

3. Serialism, parallelism, and complexity

Complexity theory deals with the classification of problems based on the *running time* and the *work space* required by algorithms for their solution. When considering parallel algorithms, we also have to take the *number of processors* into account. Complexity theory concentrates on *decision* problems (i.e., problems that produce a 'yes' or 'no' answer), but this is not a severe restriction, since most other problems can be reformulated in terms of a limited series of decision problems. An optimization problem, for example, can be solved by posing questions about the existence of a feasible solution with at most or at least a given value.

In this section, we discuss some aspects of complexity theory that are of importance to combinatorial optimization. We do not intend to go into much detail, and refer to Garey & Johnson [1979] and Cook [1981] for more complete expositions.

Sequential computers are reasonably represented by models of computation such as the Turing machine and the random access machine (RAM). Given these models, we can define several complexity classes. The class P contains the problems that are solvable in *polynomial time*, i.e., the running time is bounded by a polynomial in the problem size. The problems in P are often called *well solved* or *easy*. PSPACE contains the problems that are solvable in *polynomial space*, i.e., in work space that is bounded by a polynomial in the problem size. A very well studied class included in PSPACE is *NP*, the class of problems for which a feasible solution can be recognized as such in polynomial time. It is obvious that $P \subseteq NP \subseteq$ PSPACE, and it is conjectured that both these inclusions are proper.

Another class contained in PSPACE, which has not attracted much attention in the context of serial computations, is POLYLOGSPACE. It consists of the problems that are solvable in *polylog space*, i.e., in work space that is polynomially bounded in the logarithm of the problem size. Many problems in P belong to POLYLOGSPACE, but it is generally believed that $P \not\subseteq$ POLYLOGSPACE. We do know, however, that POLYLOGSPACE $\neq$ PSPACE.

The classes PSPACE and *NP* have their *complete* members. The PSPACE-complete problems are generalizations of all other problems in PSPACE in terms of transformations that require polynomial time. More precisely: a problem is *PSPACE-complete under polynomial-time transformations* if it belongs to PSPACE and if any other problem in PSPACE is reducible to it by a transformation that requires polynomial time. It follows that, if any PSPACE-complete problem can be shown to belong to P, then PSPACE $= P$. Since this equality is not believed to be true, a polynomial-time algorithm for a PSPACE-complete problem is very unlikely to exist. For the class *NP* and its complete members, the same properties hold.

P also has its complete problems. The P-complete problems generalize all other problems in P in terms of transformations that require logarithmic work space. Formally: a problem is *log space complete for P* or, better, *P-complete under log-space transformations*, if it belongs to P and if any other problem in P is reducible to it by a transformation using logarithmic work space. If any P-complete problem would belong to POLYLOGSPACE, then $P \subseteq$ POLYLOGSPACE. As this inclusion is believed to be false, an algorithm for a P-complete problem that uses only polylogarithmic work space cannot be expected to exist.

Serial and parallel computations are related by a hypothesis known as the *parallel computation thesis* [Chandra, Kozen & Stockmeyer, 1981; Goldschlager, 1982]: *time bounded parallel machines are polynomially related to space bounded sequential machines*. That is, for any function T of the problem size n, the class of problems solvable by a machine with unbounded parallelism in *time* $T(n)^{O(1)}$ (i.e., polynomial in $T(n)$) is equal to the class of problems solvable by a sequential machine in *space* $T(n)^{O(1)}$. This thesis is a *theorem* for many 'reasonable' parallel machine models and 'well-behaved' time bounds; see Van Emde Boas [1985] for a survey.

A frequently used model of parallel computation is the parallel random access machine, or PRAM. The PRAM is a machine with an unbounded number of processors and a shared memory. The processors perform their operations in a synchronized fashion. Simultaneous reads from the same memory location are allowed, but simultaneous writes into the same memory location are prohibited. The computation starts with one processor

activated; at any step, an active processor can do a standard operation or activate another processor; and the computation stops when the initial processor halts.

Current technology prohibits the realization of a shared memory and, hence, of a machine with PRAM-like properties. However, the PRAM model is of theoretical interest. It helps us in investigating the intrinsic parallelism in problems and algorithms. For example, Fortune & Wyllie [1978] showed that the class of problems solvable in $T(n)^{O(1)}$ time by a PRAM is equal to the class of problems solvable in $T(n)^{O(1)}$ work space by a Turing machine, if $T(n) \geq \log n$.

As a consequence, the class of problems solvable by a PRAM in polynomial time is equal to *P*SPACE. Since the PRAM is able to solve the apparently difficult problems in *P*SPACE (such as the *P*SPACE-complete and *NP*-complete ones) in polynomial time, it is obviously an extremely powerful model. The theorem by Fortune & Wyllie also implies that the problems in POLYLOGSPACE are exactly the ones solvable by a PRAM in *polylog parallel time*, i.e., in time that is polynomially bounded in the logarithm of the problem size. This leads to a distinction within the class *P*.

The problems in *P* belonging to POLYLOGSPACE are solvable in polylog parallel time. They can be considered to be among the *easiest* problems in *P*, in the sense that the influence of problem size on solution time has been limited to a minimum. (It should be noted here that a further reduction to sublogarithmic solution time is generally impossible. One reason for this is that a PRAM needs $O(\log n)$ time to activate n processors. A similar reason is that in any realistic model of parallelism a constant upper bound on the maximum number of connections of any processor to other processors leads to a logarithmic lower bound on the communication time between processors. That is, a fixed degree implies at least a logarithmic diameter of the processor network.)

On the other hand, the *P*-complete problems are unlikely to admit solution in polylog parallel time. If any such problem would be solvable in polylog parallel time, it would belong to POLYLOGSPACE, and it would follow that $P \subseteq$ POLYLOGSPACE. Hence, their solution in polylog parallel time is not expected. Any solution method for these *hardest* problems in *P* is likely to require superlogarithmic time and is therefore, loosely speaking, probably 'inherently sequential' in nature. This does not imply, of course, that parallelism cannot yield substantial speedups.

We can, therefore, distinguish within *P* between the 'very easy' problems, which are solvable in polylog parallel time, and the 'not so easy' ones, for which such a speedup due to parallelism is unlikely.

The picture of the PRAM model as sketched above is in need of some qualification. The model is theoretically very useful, but its unbounded parallelism is hardly realistic. The reader will have no difficulty in verifying that a PRAM is able to activate a superpolynomial number of processors in subpolynomial time. If a polynomial time bound is considered reasonable, then certainly a polynomial bound on the number of processors should be imposed. It is a trivial observation, however, that the class of problems solvable if both bounds are respected is simply equal to *P*. Within this more reasonable model, *NP*-complete and *P*SPACE-complete problems remain as hard as they were without parallelism.

Discussions along these lines have led to the consideration of *simultaneous resource bounds* and to the definition of new complexity classes. For example, *Nick* (*Pippenger*)*'s Class NC* contains all problems solvable in polylog parallel time on a polynomial number of processors, and *Steve* (*Cook*)*'s Class SC* contains all problems solvable in polynomial sequential time and polylog space. Some sort of extended parallel computation thesis might suggest that *NC* = *SC*. This is a major unresolved issue in complexity theory, and outside the scope of this paper. We refer to Johnson [1983] for further details and more references.

4. Local search for the TSP

In the traveling salesman problem, one is given a complete undirected graph G with vertex set $\{1, \ldots, n\}$ and a travel time d_{ij} for each edge $\{i, j\}$, and one wishes to find a Hamiltonian cycle (i.e., a cycle passing through each vertex exactly once) of minimum total duration. We assume that the travel times satisfy the triangle inequality, i.e., $d_{ij} + d_{jk} \geq d_{ik}$ for each triple (i, j, k). The TSP is a well-known *NP*-hard problem, for which many optimization and approximation algorithms have been proposed; cf. Lawler, Lenstra, Rinnooy Kan & Shmoys [1985].

We consider the following local search algorithm for the TSP. Construct an initial Hamiltonian cycle by taking an arbitrary permutation of the vertices or by applying a specific heuristic method such as the *nearest*

neighbor rule or the *double minimum spanning tree* algorithm. Then try to improve the tour by replacing a set of k of its edges by another set of k edges, and iterate until no further improvement is possible. Such replacements are called *k-exchanges*, and a tour that cannot be improved by a k-exchange is said to be *k-optimal*. We will consider the case $k = 2$ in detail. For $k > 2$, the analysis is conceptually similar but technically more involved.

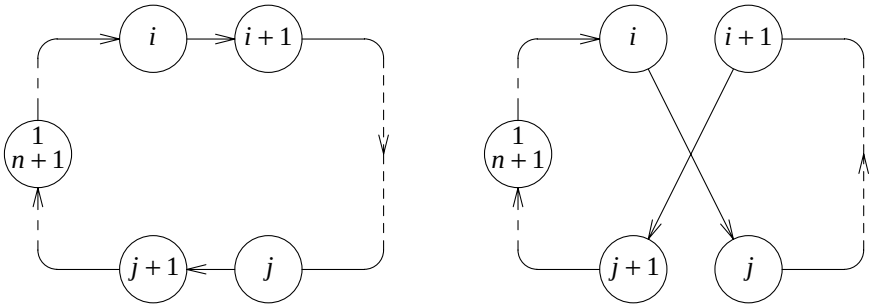

Figure 1. A 2-exchange.

For notational convenience, we consider the tour $(1, 2, \ldots, n, n+1)$, where the origin 1 and the destination $n+1$ denote the same vertex. A 2-exchange replaces two edges $\{i, i+1\}$ and $\{j, j+1\}$ of the tour by two other edges $\{i, j\}$ and $\{i+1, j+1\}$, thereby reversing the path from $i+1$ to j; see Figure 1. It is an open question if there exists a polynomial-time algorithm that obtains a 2-optimal tour by a sequence of 2-exchanges [Johnson, Papadimitriou & Yannakakis, 1988]. We therefore restrict ourselves to deciding whether a given tour is 2-optimal.

Because the travel times between the vertices do not depend on the direction, a 2-exchange results in a local improvement if and only if

$$d_{ij} + d_{i+1,j+1} < d_{i,i+1} + d_{j,j+1}.$$

Testing a single 2-exchange for improvement involves only a constant amount of information and hence requires constant time. It follows that verifying 2-optimality takes $O(n^2)$ time. No algorithm that proceeds by enumerating all possible improvements can run faster, as there are $\binom{n}{2}$ 2-exchanges.

5. Parallel local search for the TSP

Before discussing the verification of 2-optimality on the PRAM model, we will first consider an elementary problem and describe a basic technique in parallel computing for its solution.

The problem is to find the *partial sums* of a given sequence of n numbers. For the sake of simplicity, let $n = 2^m$ and suppose that the n numbers are given by $a_n, a_{n+1}, \ldots, a_{2n-1}$. We wish to find the partial sums $a_n + \cdots + a_{n+j}$ for $j = 0, \ldots, n-1$. The following procedure is due to Dekel & Sahni [1983]:

for $l \leftarrow m-1$ **downto** 0 **do**
  par $[2^l \le j \le 2^{l+1} - 1]$ $a_j \leftarrow a_{2j} + a_{2j+1}$;
$b_1 \leftarrow a_1$;
for $l \leftarrow 1$ **to** m **do**
  par $[2^l \le j \le 2^{l+1} - 1]$ $b_j \leftarrow$ **if** j odd **then** $b_{(j-1)/2}$ **else** $b_{j/2} - a_{j+1}$.

Here, a statement of the form '**par** $[\alpha \le j \le \omega]$ s_j' denotes that the statements s_j are executed in parallel for all values of j in the indicated range.

The computation is illustrated in Figure 2. In the first phase, represented by solid arrows, the sum of the a_j's is calculated. Note that the a-value corresponding to a non-leaf node is set equal to the sum of all a-values corresponding to the leaves descending from that node. In the second phase, represented by dotted arrows, each parent node sends a b-value (starting with $b_1 = a_1$) to its children: the right child receives the same value, the left one receives that value minus the a-value of the right child. The b-value of a certain node is therefore equal to the sum of all a-values of the nodes of the same generation, except those with a higher index. This implies, in

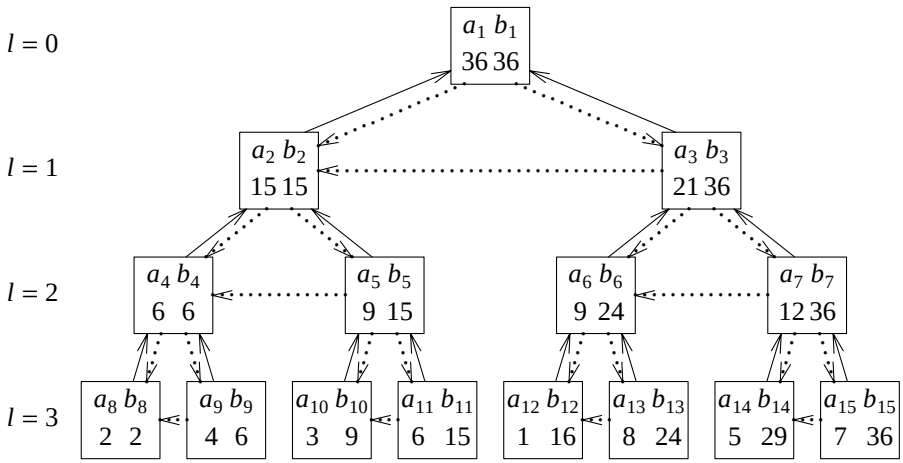

Figure 2. Partial sums: an instance with $n = 8$.

particular, that at the end we have $b_{n+j} = a_n + \cdots + a_{n+j}$ for $j = 0, \ldots, n-1$.

The algorithm requires $O(\log n)$ time and n processors. This can be improved to $O(\log n)$ time and $O(n/\log n)$ processors by a simple device. First, the set of n numbers is partitioned into $n/\log n$ groups of size $\log n$ each, and $n/\log n$ processors determine the sum of each group in the traditional serial way in $\log n$ time. After this aggregation process, the above algorithm computes the partial sums over the groups; this requires $O(n/\log n)$ processors and $O(\log n)$ time. Finally, a disaggregation process is applied with the same processor and time requirements.

In the form given above, the algorithm does not work for operations such as maximization. The partial sums algorithm uses subtraction, which has no equivalent in the case of maximization. We therefore present a version of the partial sums algorithm which is not quite so elegant as the original one, but which has the desired property since it makes use of addition only. It also runs in $O(\log n)$ time using $O(n/\log n)$ processors:

for $l \leftarrow m-1$ **downto** 0 **do**
  par $[2^l \le j \le 2^{l+1}-1]$ $a_j \leftarrow a_{2j} + a_{2j+1}$;
for $l \leftarrow 0$ **to** m **do**
  par $[2^l \le j \le 2^{l+1}-1]$
    $b_j \leftarrow$ **if** $j = 2^l$ **then** a_j **else if** j odd **then** $b_{(j-1)/2}$ **else** $b_{(j-2)/2} + a_j$.

We now return to the verification of 2-optimality. The following procedure decides whether or not the tour $(1, 2, \ldots, n, n+1)$ is 2-optimal:

par $[1 \le i < j \le n]$ $\delta_{ij} \leftarrow d_{ij} + d_{i+1,j+1} - d_{i,i+1} - d_{j,j+1}$;
$\delta_{\min} \leftarrow \min\{\delta_{ij} | 1 \le i < j \le n\}$;
if $\delta_{\min} \ge 0$
then $(1, 2, \ldots, n, n+1)$ is a 2-optimal tour
else let i^* and j^* be such that $\delta_{i^*j^*} = \delta_{\min}$,
  $(1, \ldots, i^*, j^*, j^*-1, \ldots, i^*+1, j^*+1, \ldots, n+1)$ is a shorter tour.

By adapting the first phase of the partial sums algorithm such that it computes the minimum of a set of numbers and also delivers an index for which the minimum is attained, the above procedure can be implemented to require $O(\log n)$ time and $O(n^2/\log n)$ processors. The total computational effort is $O(\log n \cdot n^2/\log n) = O(n^2)$, as it is in the serial case. This is called a *full processor utilization* or a *perfect speedup*.

Although the serial and parallel implementations seem similar, there is a basic distinction. When the tour under consideration is not 2-optimal, the serial algorithm will detect this after a number of steps that is somewhere in between 1 and $\binom{n}{2}$. In the parallel algorithm, confirmation and negation of 2-optimality always take the same amount of time.

6. Local search for the time-constrained TSP

In the TSP with time windows, each vertex i has a time window on the departure time, denoted by $[s_i, t_i]$. The time window is opened at time s_i and closed at time t_i. If the salesman arrives at i before s_i, he has to wait; if he arrives after t_i, he is late and his tour is infeasible.

Due to the presence of time windows, there are feasible and infeasible tours, and this complexifies the problem. To start with, the problem of determining the existence of a feasible tour is *NP*-complete in the strong sense. This follows from the observation that the unconstrained TSP has a tour of duration no more than B if and only if there is a feasible tour for the constrained TSP in which each vertex has a time window $[0, B]$.

Secondly, when applying local search, we have to test all candidate improvements for feasibility. A k-exchange influences the arrival times at all vertices visited after the first change in the tour. This may lead to changes in the departure times and even to infeasibility. In a straightforward implementation, we need $O(n)$ time to handle a single k-exchange, which results in a time complexity of $O(n^{k+1})$ for the verification of k-optimality. We will show how to reduce this time bound by an order n, thereby obtaining the same time complexity as in the unconstrained case.

The basic idea is the use of a specific *search strategy* in combination with a set of *global variables* such that testing the feasibility of a single exchange and maintaining the set of global variables require no more than constant time. We consider the case $k = 2$ in detail.

As before, we consider the tour $(1, 2, \ldots, n, n+1)$. We assume that this tour is feasible. A 2-exchange involves the replacement of the edges $\{i, i+1\}$ and $\{j, j+1\}$ by the edges $\{i, j\}$ and $\{i+1, j+1\}$. Such an exchange is both feasible and profitable if and only if the following three conditions are satisfied:

(1) the reversed path $(j, \ldots, i+1)$ is feasible, i.e., the new departure time at vertex k is not larger than t_k, for $k = i+1, \ldots, j$;

(2) the new departure time at vertex $j+1$ is smaller than it was before the exchange;

(3) a part of the gain at vertex $j+1$ can be carried through to the destination, i.e., the original departure time at vertex k is strictly larger than s_k, for $k = j+1, \ldots, n$.

Condition (3) needs further consideration. If it is violated, the exchange will not affect the duration of the tour. However, it will reduce the duration of the path from 1 to $k-1$, for the smallest k for which violation occurs. In the sequel, we will drop condition (3), for two reasons. First, introducing some slack may be beneficial for the rest of the procedure, even though the slack cannot be carried through to the end of the tour. In addition, taking condition (3) into account would make the presentation needlessly complicated.

We propose a *search strategy* that examines the 2-exchanges in lexicographic order. We choose i successively equal to $1, 2, \ldots, n-2$; this will be referred to as the outer loop. For a fixed value of i, we choose j successively equal to $i+2, i+3, \ldots, n$; this will be called the inner loop. In the inner loop, the previously reversed path $(j-1, \ldots, i+1)$ is repeatedly expanded with the edge $\{j, j-1\}$; cf. Figure 3.

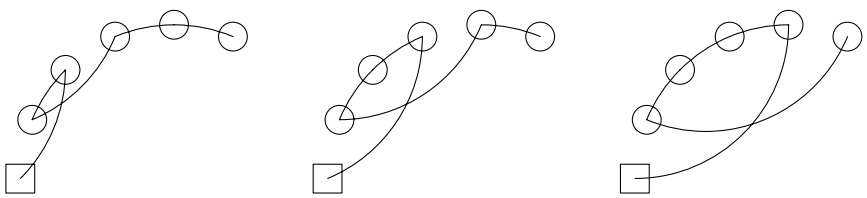

Figure 3. The search strategy for 2-exchanges.

In the following, we assume that i is fixed and consider the inner loop. The departure time at vertex k in the tour $(1, 2, \ldots, n, n+1)$ will be denoted by D_k, for $k = 1, \ldots, n+1$. The waiting and departure times at vertex k

after reversal of the path $(i+1, \ldots, j)$ will be denoted by W_k^j and D_k^j, respectively, for $k > i$.

We define three *global variables*, which will be maintained throughout the inner loop. We suppose that the reversed path $(j-1, \ldots, i+1)$ has been considered. First, T is equal to the total travel time along this path:

$T = \sum_{k=i+1}^{j-2} d_{k,k+1}$.

Secondly, W is equal to the total waiting time along the path after departing from vertex $j-1$:

$W = \sum_{k=i+1}^{j-2} W_k^{j-1}$.

Thirdly, S is equal to the maximum forward shift in time of the departure time at vertex $j-1$ that would cause no time window violation along the path:

$S = \min_{i+1 \le k \le j-1} \{t_k - (D_{j-1}^{j-1} + \sum_{l=k}^{j-2} d_{l,l+1})\}$.

Expanding the reversed path $(j-1, \ldots, i+1)$ with the edge $\{j, j-1\}$ may change the arrival time at vertex $j-1$ and thereby all departure times along the path $(j-1, \ldots, i+1)$. We define a *local variable* Δ to denote the difference between the new arrival time and the old departure time at vertex $j-1$:

$\Delta = D_j^j + d_{j,j-1} - D_{j-1}^{j-1}$.

Δ can be computed in constant time, using $D_j^j = \max\{s_j, D_i + d_{ij}\}$ and $D_{j-1}^{j-1} = \max\{s_{j-1}, D_i + d_{i,j-1}\}$.

In order to prove that we can verify 2-optimality of the tour $(1, 2, \ldots, n, n+1)$ in $O(n^2)$ time, we have to establish two facts: it is possible to update the values of the global variables in constant time, and the new values allow us to handle a single 2-exchange in constant time.

As to updating the global variables, we note that the definition of Δ covers two cases. In the case that $\Delta < 0$, the triangle inequality implies that the old arrival at $j-1$ cannot have been later than the new arrival. It follows that the old arrival and departure times did not coincide, so that the old departure occurred at the opening of the time window. But then we have that $-\Delta = W_{j-1}^j$, the new waiting time at $j-1$. In the case that $\Delta \ge 0$, we obviously have $\Delta = D_{j-1}^j - D_{j-1}^{j-1}$, the forward shift of the departure time at $j-1$. We conclude that the new values of the global variables are obtained by

$T \leftarrow T + d_{j-1,j}$,

$W \leftarrow \max\{W - \Delta, 0\}$,

$S \leftarrow \min\{t_j - D_j^j, S - \Delta\}$.

These updates require constant time.

As to handling a single 2-exchange, the conditions (1), requiring feasibility, and (2), stipulating profitability at vertex $j+1$, can be written as

(1) $D_k^j \le t_k$ for $k = i+1, \ldots, j$,

(2) $D_{j+1}^j < D_{j+1}$.

The inequalities (1) are obviously equivalent to $S \ge 0$; see Savelsbergh [1988] for a formal proof. For inequality (2), we observe that the new departure time at $j+1$ satisfies

$D_{j+1}^j = \max\{s_{j+1}, D_j^j + T + W + d_{i+1,j+1}\}$.

We conclude that conditions (1) and (2) can be tested in constant time.

7. Parallel local search for the time-constrained TSP

We will now present a parallel algorithm for verifying 2-optimality of a time-constrained TSP tour. It requires $O(\log n)$ time and $O(n^2/\log n)$ processors, and thereby has the same resource requirements as in the unconstrained case.

Again, we consider the tour $(1, 2, \ldots, n, n+1)$, which is assumed to be feasible. We start by computing all partial path lengths along the tour. This enables us to construct the tours that can be obtained by a 2-exchange.

Our algorithm has five phases.

(1) We first compute all partial sums T_{ij} of travel times along the tour:

par $[1 \le i \le j \le n+1]$ $T_{ij} \leftarrow \sum_{k=i}^{j-1} d_{k,k+1}$.

By application of the partial sums algorithm from Section 5, this phase requires $O(\log n)$ time and $O(n^2/\log n)$ processors.

(2) We now investigate the effect of the time windows on the paths along the tour. For each pair of vertices $\{i, j\}$ with $i < j$, we define E_{ij} as the earliest possible departure time at vertex j when traveling along the tour from i to j, and E_{ji} as the earliest possible departure time at vertex i when traveling from j to i in the reverse direction along the tour. Note that $E_{1,n+1}$ is the arrival time at vertex 1. Further, let L_{ij} denote the latest possible departure time at vertex i such that the path from i to j remains feasible, and let L_{ji} denote the latest possible departure time at vertex j such that the path from j to i remains feasible. We then have:

par $[1 \le i \le j \le n+1]$ $E_{ij} \leftarrow \max_{i \le k \le j}(s_k + T_{kj})$;
par $[1 \le i \le j \le n+1]$ $E_{ji} \leftarrow \max_{i \le k \le j}(s_k + T_{ik})$;
par $[1 \le i \le j \le n+1]$ $L_{ij} \leftarrow \min_{i \le k \le j}($ **if** $E_{ik} \le t_k$ **then** $t_k - T_{ik}$ **else** $-\infty)$;
par $[1 \le i \le j \le n+1]$ $L_{ji} \leftarrow \min_{i \le k \le j}($ **if** $E_{jk} \le t_k$ **then** $t_k - T_{kj}$ **else** $-\infty)$.

Using the partial sums algorithm from Section 5 with addition replaced by maximization or minimization, we have the same time and processor requirements as in phase (1).

(3) Given the earliest and latest possible departure times relative to paths along the tour, we compute the earliest departure time $D_{ij}(k)$ at any vertex k and the earliest arrival time A_{ij} at the origin after the replacement of the edges $\{i, i+1\}$ and $\{j, j+1\}$ by the edges $\{i, j\}$ and $\{i+1, j+1\}$:

par $[1 \le i < j \le n]$ $D_{ij}(j) \leftarrow \max\{E_{1i} + d_{ij}, s_j\}$;
par $[1 \le i < j \le n]$ $D_{ij}(i+1) \leftarrow \max\{D_{ij}(j) + T_{i+1,j}, E_{j,i+1}\}$;
par $[1 \le i < j \le n]$ $D_{ij}(j+1) \leftarrow \max\{D_{ij}(i+1) + d_{i+1,j+1}, s_{j+1}\}$;
par $[1 \le i < j \le n]$ $A_{ij} \leftarrow \max\{D_{ij}(j+1) + T_{j+1,n+1}, E_{j+1,n+1}\}$.

For this phase we need $O(1)$ time and $O(n^2)$ processors, or $O(\log n)$ time and $O(n^2/\log n)$ processors.

(4) We then test for the feasibility of the tours obtained by 2-exchanges, using boolean variables F_{ij}:

par $[1 \le i < j \le n]$ $F_{ij} \leftarrow (D_{ij}(j) \le L_{j,i+1}) \,\&\, (D_{ij}(j+1) \le L_{j+1,n+1})$.

The first condition tests for feasibility at the vertices $i+1, \ldots, j$ and the second one at the vertices $j+1, \ldots, n+1$. As in the previous phase, we need $O(1)$ time and $O(n^2)$ processors, or $O(\log n)$ time and $O(n^2/\log n)$ processors.

(5) Finally, we decide whether or not the given tour is 2-optimal in the same way as in the case without time windows:

$A_{\min} \leftarrow \min\{A_{ij} | F_{ij}, 1 \le i < j \le n\}$;
if $E_{1,n+1} \le A_{\min}$
then $(1, 2, \ldots, n, n+1)$ is a 2-optimal tour
else let i^* and j^* be such that $F_{i^*j^*}$ & $A_{i^*j^*} = A_{\min}$,
$(1, \ldots, i^*, j^*, j^*-1, \ldots, i^*+1, j^*+1, \ldots, n+1)$ is a better feasible tour.

For this last phase, the same time and processor bounds as before suffice. So, we end up with an algorithm that runs in $O(\log n)$ time using $O(n^2/\log n)$ processors, which is the same as in the case without time windows.

For each fixed $k > 2$, we can derive a logarithmic-time algorithm along similar lines. One has to take into account that, given k edges, several k-exchanges are possible. Further, the influence of a k-exchange on a tour is more complex. However, it is not hard to see that the running time remains $O(\log n)$ using $O(n^k/\log n)$ processors, which is optimal with respect to the number $\Theta(n^k)$ of k-exchanges.

References

J.M. Anthonisse, J.K. Lenstra, M.W.P. Savelsbergh (1989). Behind the screen: DSS from an OR point of view. *Decision Support Syst. 4*, 413-419.

M. Bellmore, G.L. Nemhauser (1968). The traveling salesman problem: a survey. *Oper. Res. 16*, 538-558.

A.K. Chandra, D.C. Kozen, L.J. Stockmeyer (1981). Alternation. *J. Assoc. Comput. Mach. 28*, 114-133.

S.A. Cook (1981). Towards a complexity theory of synchronous parallel computation. *Enseign. Math. (2) 27*, 99-124.

E. Dekel, S. Sahni (1983). Binary trees and parallel scheduling algorithms. *IEEE Trans. Comput. C-32*, 307-315.

S. Fortune, J. Wyllie (1978). Parallelism in random access machines. *Proc. 10th Annual ACM Symp. Theory of Computing*, 114-118.

M.R. Garey, D.S. Johnson (1979). *Computers and Intractability: a Guide to the Theory of NP-Completeness*, Freeman, San Francisco.

L.M. Goldschlager (1982). A universal connection pattern for parallel computers. *J. Assoc. Comput. Mach. 29*, 1073-1086.

D.S. Johnson (1983). The NP-completeness column: an ongoing guide; seventh edition. *J. Algorithms 4*, 189-203.

D.S. Johnson, C.H. Papadimitriou, M. Yannakakis (1988). How easy is local search? *J. Comput. Syst. Sci. 37*, 79-100.

R. Jonker (1986). *Traveling Salesman and Assignment Algorithms: Design and Implementation*, Ph.D. thesis, University of Amsterdam.

R. Jonker, G. de Leve, J.A. van der Velde, A. Volgenant (1980). Rounding [Bounding] symmetric traveling salesman problems with an asymmetric assignment problem. *Oper. Res. 28*, 623-627.

G.A.P. Kindervater (1989). *Exercises in Parallel Combinatorial Computing*, Ph.D. thesis, Centre for Mathematics and Computer Science, Amsterdam.

E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys (eds.) (1985). *The Traveling Salesman Problem: a Guided Tour of Combinatorial Optimization*, Wiley, Chichester.

S. Lin (1965). Computer solutions of the traveling salesman problem. *Bell System Tech. J. 44*, 2245-2269.

M.W.P. Savelsbergh (1988). *Computer Aided Routing*, Ph.D. thesis, Centre for Mathematics and Computer Science, Amsterdam.

R. Tijdeman (1968). *Het Handelsreizigersprobleem, een Literatuuronderzoek*, Report S385, Mathematical Centre, Amsterdam.

P. van Emde Boas (1985). The second machine class: models of parallelism. J. van Leeuwen, J.K. Lenstra (eds.). *Parallel Computers and Computations*, CWI Syllabus 9, Centre for Mathematics and Computer Science, Amsterdam, 133-161.

K.M. van Hee (1989). Private communication.

A. Volgenant (1987). *Contributions to the Solution of the Traveling Salesman Problem and Related Problems*, Ph.D. thesis, University of Amsterdam.